Type-directed search with dependent types

Ben Sherman

August 12, 2014

Overview

- Type systems
- Code & type search
- Equality and isomorphism
- *:search* in Idris

Expressiveness of type systems

Inexpressive

- Python
- C
- ML
- Haskell
- Idris, Agda

Expressive

Python

- Untyped: can't determine anything important statically
- There are
  - Objects: $*$
  - n-ary functions on objects: $*^n \to *$

C

"What's the worst a function can do that takes a **void** ∗ and returns a **void** ∗?"

```
void * id(void * x) {
  strcpy((char *) x, "Bye, bye, data!");
  strcpy((char *) &x, "Bye, bye, stack!");
  return (void *) rand();
}
```

ML

"With parametric polymorphism, *id* can only be one thing!"

```
val id = (fn x => (
  print "Starting evil ...";
  (* Doing evil ... *)
  print "Finishing evil ...";
  x)) : ('a -> 'a);
```

Haskell

"In a purely functional language, *id* can only be one thing!"

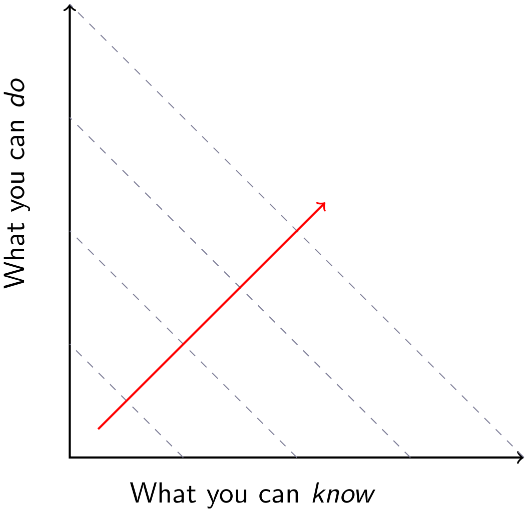
```
id :: a -> a
id = id
```

Idris

In a *total* language, we finally win!

```
total
id : (a : Type) -> a -> a
id _ x = x
```

Programming language power



Test-driven development

http://math.stackexchange.com/questions/111440/examples-of-apparent-patterns-that-eventually-fail?lq=1

Termination checking with tests?

The busy beaver function

0	0
1	1
2	6
3	21
4	107
5	$> 47,176,870$
6	$> 10^{36534}$

Proving map fusion

The Curry-Howard correspondence

Haskell	Logic
type variables : a	proposition variables : p
types : Bool	propositions : "Socrates is a man"
function types : a -> b	implications (implies) : $p \to q$
tuples : (a, b)	conjunctions (and) : $p \wedge q$
either : Either a b	disjunctions (or) : $p \vee q$
type inhabitation : id :: a -> a	truth : $\vdash p \to p$

The type is the *what*. The value is the *why*.

For any positive integers n, x, y and z where n is greater than 2, $x^n + y^n \neq z^n$.

$$\forall n, x, y, z \in \mathbb{N}.$$

$$n > 2, x > 0, y > 0, z > 0 \quad \rightarrow \quad x^n + y^n \neq z^n$$

```
(n, x, y, z : Nat) →
        n > 2 → x > 0 → y > 0 → z > 0
                          →      Not (x^n + y^n = z^n)
```

Sorting a list

Haskell:

```
sort :: Ord a ⇒ [a] → [a]
```

Idris (my example, > 150 LOC):

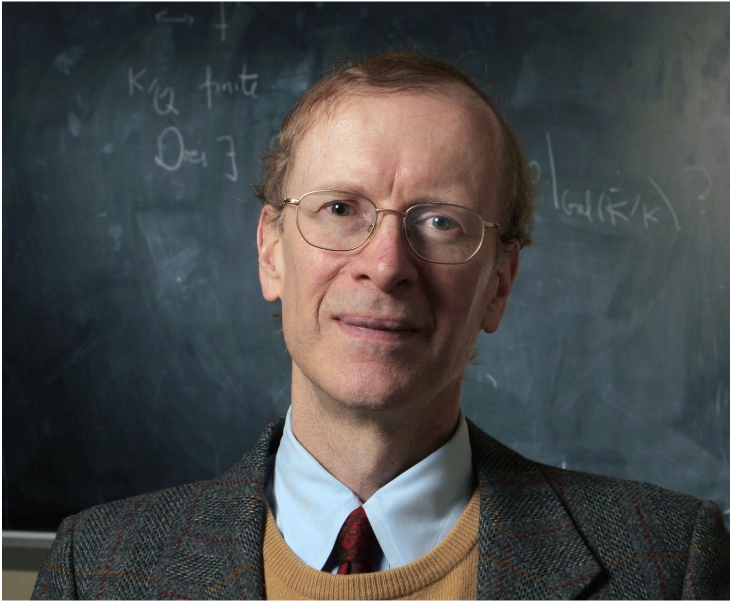
```
quickSort : {a : Type} → {less : a → a → Type}
  → {eq : a → a → Type}
  → TotalOrder less eq
  → (xs : List a)
  → Exists (List a) (\ys ⇒ (IsSorted less ys, Permutation xs ys))
```

Type-driven development

Types

- Prove properties stronger than any test can show
- Are documentation that is never wrong or outdated
- Provide an exact specification

Type-driven development



Why code search matters

- Stand on the shoulders of giants
  - Modern software development heavily depends on library re-use
- Number of libraries increasing drastically
- Code size of projects increasing drastically
- (Purely) functional programming is the modular solution for scaling to large systems

Search difficulties

- “Haskell stack overflow”, “Go tree”, “Go map”
- Ord (Haskell) vs. Comparable (Java)
    - (In Java, all identifiers must have at least 8 characters?)

Search

What's in a name? that which we call a rose

By any other name would smell as sweet;

William Shakespeare, *Romeo and Juliet*

Type-directed search

- Can choose your name; can't choose your type!
- *Semantics* instead of names
- Tool of choice for type-driven developers

Hoogle

- Type-directed search for Haskell
- 2000 searches per day (2011)
- Based on a notion of edit distance

Hoogle mutations

Aliases	$\text{String} \longleftrightarrow [\text{Char}]$
Instances	$\text{Ord a} \Rightarrow \text{a} \longleftrightarrow \text{a}$
Subtyping	$\text{Num a} \Rightarrow \text{a} \longleftrightarrow \text{Int}$
"Boxing"	$\text{a} \longleftrightarrow \text{Applicative f} \Rightarrow \text{f a}$
Free variable duplication	$(\text{a}, \text{b}) \longleftrightarrow (\text{a}, \text{a})$
Restriction	$\text{m a} \longleftrightarrow [\text{a}]$
Argument deletion	$\text{a} \to \text{b} \to \text{c} \longleftrightarrow \text{b} \to \text{c}$
Argument reordering	

Distinction without a difference

- Even though $a \to b \to c$ and $(a,\ b) \to c$ are distinct types, they "mean the same thing."
- When we search one type, we'd like to match both!
- What can we use to capture this notion?

Type isomorphism

Definition

Types A and B are *isomorphic* if there are functions $f : A \to B$ and $g : B \to A$ such that $(x : A) \to (g \circ f)(x) = x$ and $(y : B) \to (f \circ g)(y) = y$, and we write $A \cong B$.

Proposition

Isomorphism ($\cong$) is an equivalence relation.

(Type *equivalence* in HoTT)

What does $=$ mean?

Notions of equality

- In Haskell, not all types allow their terms to be compared for equality (e.g., IO ())
- In Idris, in order to perform type checking, for any arbitrary type, we must be able to compare terms of that type for equality!

Equality in Idris

- Definitional equality, $\equiv$, for when terms are "obviously" equal
  - Used for type checking
- Propositional equality
  - $(=) : (x : A) \to (y : B) \to \text{Type } \textbf{where}$
  - $\text{refl} : \{A : \text{Type}\} \to \{x : A\} \to x = x$
- Transport
  - $\text{replace} : a = b \to P\ a \to P\ b$

Equality of functions

Axiom of function extensionality:

```
funext : (f, g : a → b) -> ((x : a) → f x = g x) → f = g
```

Type isomorphism

Proposition

If types $A \cong B$, and $t : Type \vdash M : Type$, then $[A/t]M \cong [B/t]M$.

Proof.

Suppose we have $p : [A/t]M$ and want $q : [B/t]M$. Intuitively, when we need to produce a B in q, we use code from p to make an A and then map it to B. When we must use a B in a, we map it to A and then use code from p to use that value. $\square$

Type isomorphism

Type isomorphism is similar to set bijection:

Proposition

If there is some $n \in \mathbb{N}$ such that A and B each have n elements, then A and B are isomorphic.

Proof.

Construct isomorphisms from A to $\mathrm{Fin}\ n$ and B to $\mathrm{Fin}\ n$. Since $\cong$ is an equivalence relation, $A \cong \mathrm{Fin}\ n \cong B$. $\square$

Type isomorphism in Haskell

```
A = X → Y → Z
B = (X, Y) → Z
f = uncurry
g = curry
```

Decidability of isomorphism

Proposition

Type isomorphism is undecidable in System F (Haskell) and intuitionistic type theory (Idris).

Proof.

- *Claim:* A type is isomorphic to $\perp$ if and only if it is uninhabited.
- Type inhabitation is undecidable in System F and intuitionistic type theory.

$\square$

Another notion of isomorphism

$$\frac{x = y}{x \cong y}$$

$$\frac{\begin{array}{l} f : A \to B \\ g : B \to A \\ \_ : (x : A) \to (g \circ f)(x) \cong x \\ \_ : (y : B) \to (f \circ g)(y) \cong y \end{array}}{A \cong B}$$

Isomorphism is not enough!

Suppose we want to compare two values whose type has instance Ord for equality. We search

$$\mathrm{Ord}\ a \Rightarrow a \to a \to \mathrm{Bool}$$

We'd like to find

$$(==) :: \mathrm{Eq}\ a \Rightarrow a \to a \to \mathrm{Bool}$$

Its type is strictly *more* general than what we asked for.

Isomorphism is not enough!

```
Ord a ⇒ a → a → Bool
```

If we take this too far, though, results may not be useful:

```
const (const True) :: a → a → Bool
```

Too general!

Type containment

We want a partial order $\succeq$ that defines isomorphism: that is,

If $A \succeq B$ and $B \succeq A$, then $A \cong B$.

A first pass at type containment

Definition

Type A *covers* B if there is a subset $A' \subseteq A$ and functions $f : A' \to B$ and $g : B \to A'$ such that $g \circ f = \mathrm{id}_{A'}$ and $f \circ g = \mathrm{id}_B$, and we write $A \succeq B$.

Proposition

If $A \succeq B$ and $B \succeq A$, then $A \cong B$.

Proof.

Myhill isomorphism theorem? □

Strategy for defining type containment

- Define a partial order $\succeq$ on types such that the resulting equivalence relation $\cong$ is *sound* with respect to isomorphism
  - *sound*: If $A \cong B$, then A is isomorphic to B
  - But if A is isomorphic to B, no guarantee of *any* relation between A and B

A definition of type containment in Haskell

- Type instantiation (with a concrete type)
  - $\text{Maybe a} \quad \succ \quad \text{Maybe Int}$
  - $\text{Show a} \Rightarrow \text{a} \to \text{String} \quad \succ \quad \text{Bool} \to \text{String}$
- Swapping argument order
  - $A \to B \to C \quad \cong \quad B \to A \to C$
- "Inlining" non-recursive types which have a single constructor
  - $\textbf{data}\ (,)\ \text{a b}\ \textbf{where}\ (,) :: \text{a} \to \text{b} \to (\text{a, b})$
  - $(\text{a, b}) \to \text{c} \quad \cong \quad \text{a} \to \text{b} \to \text{c}$

Canonical forms

- Take advantage of structural properties
  - $A_1 \to \cdots \to A_n \to B$ becomes $\{A_1, \ldots, A_n\} \to B$, where $\{\cdot\}$ represents a multiset.
  - Reduce complexity of comparing arguments from $n!$ to $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$
  - Similar for products (i.e. n-tuples) and sums (e.g., nested Eithers)

Towards dependent types

- Type isomorphism-based searched is *most* valuable in a language like Idris; the types are so informative!
- Closely tied to automated theorem proving, automatic program synthesis

Towards dependent types

Possible issues:

- Distinct type variables may be *dependent* on one another!
  - $(a : \text{Type}) \to (x : a) \to x = x$
  - Can't always swap argument order!
    - $(n : \text{Nat}) \to (\_ : \text{Fin } n) \to \text{Fin } (S\ n)$
- Functions in type signatures not always bijective
  - $\text{fromList} : (l : \text{List } a) \to \text{Vect } (\text{length } l)\ a$
  - $(l : \text{List } a) \to \text{Vect } (\text{length } l)\ a \succeq^{?} \text{Vect } 10\ a$
- Pervasive use of implicit arguments

Matching types

1. $\mathrm{fact}\ \ 5 = 100$
2. $120 = 100$

No results!

Matching types

1. $\text{fact}\ 5 = 120$
2. $120 = 120$
3. $(\mathrm{n} : \mathrm{Nat}) \to \mathrm{n} = \mathrm{n}$
4. $(\mathrm{t} : \mathrm{Type}) \to (\mathrm{n} : \mathrm{t}) \to \mathrm{n} = \mathrm{n}$

```
1 refl : x = x
```

Matching types

1. $(\mathrm{Ord}\ a, \mathrm{Ord}\ b, \mathrm{Eq}\ c) \Rightarrow ((a, b), c) \to ((a, b), c) \to \mathrm{Bool}$
2. $(\mathrm{Ord}\ a, \mathrm{Eq}\ b, \mathrm{Eq}\ c) \Rightarrow ((a, b), c) \to ((a, b), c) \to \mathrm{Bool}$
3. $(\mathrm{Eq}\ a, \mathrm{Eq}\ b, \mathrm{Eq}\ c) \Rightarrow ((a, b), c) \to ((a, b), c) \to \mathrm{Bool}$
4. $(\mathrm{Eq}\ (a, b), \mathrm{Eq}\ c) \Rightarrow ((a, b), c) \to ((a, b), c) \to \mathrm{Bool}$
5. $\mathrm{Eq}\ ((a, b), c) \Rightarrow ((a, b), c) \to ((a, b), c) \to \mathrm{Bool}$
6. $\mathrm{Eq}\ t \Rightarrow t \to t \to \mathrm{Bool}$

Demo

Beating Hoogle

Instant is off | Manual | haskell.org

Hoogλe

(Ord a, Ord b) => (a, b) -> (a, b) -> Bc [Search]

(Ord a, Ord b) => (a, b) -> (a, b) -> Bool

Packages

- fgl
- OpenGL

equal :: (Eq a, Eq b, Graph gr) => gr a b -> gr a b -> Bool

fgl Data.Graph.Inductive.Graph

WeightedProperties :: (GLfloat, v) -> (GLfloat, v) -> (GLfloat, v) -> (GLfloat, v) -> WeightedProperties v

OpenGL Graphics.Rendering.OpenGL.GLU.Tessellation

Triangle :: (TriangleVertex v) -> (TriangleVertex v) -> (TriangleVertex v) -> Triangle v

OpenGL Graphics.Rendering.OpenGL.GLU.Tessellation

“Kind” search for free

Instant is off | Manual | haskell.org

Hoogλe

`* -> *` Search

`* -> *`

Parse error: (line 1, column 2): unexpected " " expecting letter

For information on what queries should look like, see the user manual.

The algorithm

Roughly 4 stages:

1. Match the return type
2. Match the argument types
3. Introduce (eliminate) a subset of the typeclasses
4. Match the typeclasses

The state

Current (possibly altered) forms of:

- Arguments yet to be resolved for the left type and right type
- Typeclass constraints yet to be resolved for the left type and right type
- A record of the types of transformations which have been done so far (for keeping score)

The state transition machine

- For each type, $\mathrm{nextSteps} :: \mathrm{State} \to [\mathrm{State}]$
- $\mathrm{isFinal} :: \mathrm{State} -> \mathrm{Bool}$ tells us when we are done
- "two-level" Dijkstra's algorithm:
  1. Which type should I be working on right now?
  2. Which state should I call $\mathrm{nextSteps}$ on?

Matching arguments

- Construct a directed acyclic graph representing the argument dependencies
- Try matching one argument from each type (with unification), only considering arguments which don't appear in the types of other arguments
- Make sense of the unification result ($\mathrm{a} \sim \mathrm{f}\ \mathrm{b}$), remove variables which are completely determined, and convert the types in the appropriate places
- Repeat until all arguments are matched

Matching typeclasses

- Try to match a typeclass constraint from one type with a constraint from the other
- If there are no such matches, then try replacing a typeclass constraint with an instance, as long as the instance doesn't introduce new variables

Possible improvements

- Produce the corresponding “data” for the search results
- Inlining non-recursive datatypes
- Find isomorphic datatypes
- Bake in usage of the $\mathrm{I{\scriptsize SO}}$ typeclass
  - A safe way to make :search automatically user-extensible!
- Find an admissible heuristic for type matching scores and use A*
- Be less hacky with typeclasses

Pi in the sky

- Big database of libraries (with code that feels like programs and code that feels like proofs)
- Type-driven development
- Search the types you must implement; if there's a result, use the library with confidence that it meets the specification